Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L1	42	(orphan with object)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:43
L2	0	l1 and (directory with publish with (object or module))	USPAT; EPO; DERWENT	OR	ON	2005/01/23 10:07
L3	22	l1 and (directory)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:40
L4	1	l1 and (directory) and (status with (deleat$4 or remov$4 or prun$4))	USPAT; EPO; DERWENT	OR	ON	2005/01/23 10:27
L5	1	"5692129".pn. and (policy or schema or strategy)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 10:29
L6	1	l3 and (thread) and monitor$4	USPAT; EPO; DERWENT	OR	ON	2005/01/23 10:34
L7	46	"ADST" or (active near directory near service near interface)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 11:05
L8	35	"ADST" or (active near directory near service near interface) and (printer with object)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:40
L9	0	l8 and orphaned	USPAT; EPO; DERWENT	OR	ON	2005/01/23 11:06
L10	0	l8 and orphan$4	USPAT; EPO; DERWENT	OR	ON	2005/01/23 11:06
L11	1	l8 and thread	USPAT; EPO; DERWENT	OR	ON	2005/01/23 11:08
L12	1	l3 and thread$4 and printer	USPAT; EPO; DERWENT	OR	ON	2005/01/23 11:13
L13	34	"ADST" or (active near directory near service near interface) and (printer with object) and (thread with priority with orphan)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:41
L14	34	"ADST" or (active near directory near service near interface) and (printer with object) and (thread with low with priority with orphan)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:41
L15	0	(orphan with object) and (thread with low with priority with orphan)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:44

L16	0	(thread with low with priority with orphan)	USPAT; EPO; DERWENT	OR	ON	2005/01/23 15:44

Terms used **orphan thread low priority** Found **18** of **148,786**

Sort results by relevance Save results to a Binder

Display results expanded form Search Tips

Open results in a new window

Try an Advanced Search

Try this search in The ACM Guide

Results 1 - 18 of 18

Relevance scale

1 Enhancing software reliability with speculative threads

Jeffrey Oplinger, Monica S. Lam

October 2002 **Proceedings of the 10th international conference on Architectural support for programming languages and operating systems**, Volume 37 , 36 , 30 Issue 10 , 5 , 5

Full text available: pdf(1.47 MB) Additional Information: full citation, abstract, references, citings

This paper advocates the use of a monitor-and-recover programming paradigm to enhance the reliability of software, and proposes an architectural design that allows software and hardware to cooperate in making this paradigm more efficient and easier to program.We propose that programmers write monitoring functions assuming simple sequential execution semantics. Our architecture speeds up the computation by executing the monitoring functions speculatively in parallel with the main computation. For ...

2 A resource management framework for priority-based physical-memory allocation

Kingsley Cheung, Gernot Heiser

January 2002 **Australian Computer Science Communications , Proceedings of the seventh Asia-Pacific conference on Computer systems architecture - Volume 6**, Volume 24 Issue 3

Full text available: pdf(1.32 MB) Additional Information: full citation, abstract, references, index terms

Most multitasking operating systems support scheduling priorities in order to ensure that processor time is allocated to important or time-critical processes in preference to less important ones. Ideally this would prevent a low-priority process from slowing the execution of a high-priority one. In practice, strict prioritisation is undermined by a lack of suitable allocation policy for resources other than CPU time. For example, a low priority process may degrade the execution speed of a high-p ...

3 Distributed systems - programming and management: On remote procedure call

Patrícia Gomes Soares

November 1992 **Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research - Volume 2**

Full text available: pdf(4.52 MB) Additional Information: full citation, abstract, references, citings

The Remote Procedure Call (RPC) paradigm is reviewed. The concept is described, along with the backbone structure of the mechanisms that support it. An overview of works in supporting these mechanisms is discussed. Extensions to the paradigm that have been proposed to enlarge its suitability, are studied. The main contributions of this paper are a standard view and classification of RPC mechanisms according to different perspectives, and

a snapshot of the paradigm in use today and of goals for t ...

4 Programming languages for distributed computing systems
Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum
September 1989 **ACM C mputing Surveys (CSUR)**, Volume 21 Issue 3

Full text available: pdf(6.50 MB) Additional Information: full citation, abstract, references, citings, index terms, review

When distributed systems first appeared, they were programmed in traditional sequential languages, usually with the addition of a few library procedures for sending and receiving messages. As distributed applications became more commonplace and more sophisticated, this ad hoc approach became less satisfactory. Researchers all over the world began designing new programming languages specifically for implementing distributed applications. These languages and their history, their underlying pr ...

5 Implementation of Argus
B. Liskov, D. Curtis, P. Johnson, R. Scheifer
November 1987 **ACM SIGOPS Operating Systems Review , Proceedings of the eleventh ACM Symposium on Operating systems principles**, Volume 21 Issue 5

Full text available: pdf(1.34 MB) Additional Information: full citation, abstract, references, citings, index terms

Argus is a programming language and system developed to support the construction and execution of distributed programs. This paper describes the implementation of Argus, with particular emphasis on the way we implement atomic actions, because this is where Argus differs most from other implemented systems. The paper also discusses the performance of Argus. The cost of actions is quite reasonable, indicating that action systems like Argus are practical.

6 Fast detection of communication patterns in distributed executions
Thomas Kunz, Michiel F. H. Seuren
November 1997 **Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research**

Full text available: pdf(4.21 MB) Additional Information: full citation, abstract, references, index terms

Understanding distributed applications is a tedious and difficult task. Visualizations based on process-time diagrams are often used to obtain a better understanding of the execution of the application. The visualization tool we use is Poet, an event tracer developed at the University of Waterloo. However, these diagrams are often very complex and do not provide the user with the desired overview of the application. In our experience, such tools display repeated occurrences of non-trivial commun ...

7 A polylog time wait-free construction for closed objects
Tushar Deepak Chandra, Prasad Jayanti, King Tan
June 1998 **Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing**

Full text available: pdf(1.34 MB) Additional Information: full citation, references, citings, index terms

8 Partitioned garbage collection of a large object store
Umesh Maheshwari, Barbara Liskov
June 1997 **ACM SIGMOD Rec rd , Proceedings f the 1997 ACM SIGMOD internati nal conference n Management f data**, Volume 26 Issue 2

Full text available: pdf(1.37 MB) Additional Information: full citation, abstract, references, citings, index terms

We present new techniques for efficient garbage collection in a large persistent object store. The store is divided into partitions that are collected independently using information about inter-partition references. This information is maintained on disk so that it can be recovered after a crash. We use new techniques to organize and update this information while avoiding disk accesses. We also present a new global marking scheme to collect cyclic garbage across partitions. Global marking ...

Keywords: cyclic garbage, garbage collection, object database, partitions

9 GUM: a portable parallel implementation of Haskell

P. W. Trinder, K. Hammond, J. S. Mattson, A. S. Partridge, S. L. Peyton Jones

May 1996 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation**, Volume 31 Issue 5

Full text available: pdf(1.14 MB) Additional Information: full citation, abstract, references, citings, index terms

GUM is a portable, parallel implementation of the Haskell functional language. Despite sustained research interest in parallel functional programming, GUM is one of the first such systems to be made publicly available.GUM is message-based, and portability is facilitated by using the PVM communications harness that is available on many multi-processors. As a result, GUM is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architec ...

10 Providing high availability using lazy replication

Rivka Ladin, Barbara Liskov, Liuba Shrira, Sanjay Ghemawat

November 1992 **ACM Transactions on Computer Systems (TOCS)**, Volume 10 Issue 4

Full text available: pdf(2.46 MB) Additional Information: full citation, abstract, references, citings, index terms, review

To provide high availability for services such as mail or bulletin boards, data must be replicated. One way to guarantee consistency of replicated data is to force service operations to occur in the same order at all sites, but this approach is expensive. For some applications a weaker causal operation order can preserve consistency while providing better performance. This paper describes a new way of implementing causal operations. Our technique also supports two other kinds of operations: ...

Keywords: client/server architecture, fault tolerance, group communication, high availability, operation ordering, replication, scalability, semantics of application

11 Transactional memory: architectural support for lock-free data structures

Maurice Herlihy, J. Eliot B. Moss

May 1993 **ACM SIGARCH Computer Architecture News , Proceedings of the 20th annual international symposium on Computer architecture**, Volume 21 Issue 2

Full text available: pdf(1.10 MB) Additional Information: full citation, abstract, references, citings, index terms

A shared data structure is lock-free if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock. This paper introduces transactional memory

12 Understanding the limitations of causally and totally ordered communication

David R. Cheriton, Dale Skeen

December 1993 **ACM SIGOPS Operating Systems Review , Pr ceedings f the**

f urteenth ACM symp sium n Operating systems principles, Volume 27 Issue 5

Full text available: pdf(1.71 MB) Additional Information: full citation, abstract, references, citings, index terms

Causally and totally ordered communication support (CATOCS) has been proposed as important to provide as part of the basic building blocks for constructing reliable distributed systems. In this paper, we identify four major limitations to CATOCS, investigate the applicability of CATOCS to several classes of distributed applications in light of these limitations, and the potential impact of these facilities on communication scalability and robustness. From this investigation, we find limited meri ...

13 Fault tolerance under UNIX

Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, Wolfgang Oberle
January 1989 **ACM Transactions on Computer Systems (TOCS)**, Volume 7 Issue 1

Full text available: pdf(1.97 MB) Additional Information: full citation, abstract, references, citings, index terms, review

The initial design for a distributed, fault-tolerant version of UNIX based on three-way atomic message transmission was presented in an earlier paper [3]. The implementation effort then moved from Auragen Systems1 to Nixdorf Computer where it was completed. This paper describes the working system, now known as the TARGON/32. The original design left open questions in at least two areas: fault tolerance for server processes and recovery after a crash were brie ...

14 Industry/government track papers: TiVo: making show recommendations using a distributed collaborative filtering architecture

Kamal Ali, Wijnand van Stam
August 2004 **Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining**

Full text available: pdf(810.92 KB) Additional Information: full citation, abstract, references, index terms

We describe the TiVo television show collaborative recommendation system which has been fielded in over one million TiVo clients for four years. Over this install base, TiVo currently has approximately 100 million ratings by users over approximately 30,000 distinct TV shows and movies. TiVo uses an item-item (show to show) form of collaborative filtering which obviates the need to keep any persistent memory of each user's viewing preferences at the TiVo server. Taking advantage of TiVo's client- ...

Keywords: clustering clickstreams, collaborative-filtering

15 The Turing programming language

Richard C. Holt, James R. Cordy
December 1988 **Communications of the ACM**, Volume 31 Issue 12

Full text available: pdf(1.59 MB) Additional Information: full citation, abstract, references, citings, index terms, review

Turing, a new general purpose programming language, is designed to have Basic's clean interactive syntax, Pascal's elegance, and C's flexibility.

16 File and storage systems: The Google file system

Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung
October 2003 **Proceedings f the nineteenth ACM symposium n Operating systems principles**

Full text available: pdf(275.54 KB) Additional Information: full citation, references, index terms

Keyw rds: clustered storage, data storage, fault tolerance, scalability

17 NSF workshop on industrial/academic cooperation in database systems
Mike Carey, Len Seligman
March 1999 **ACM SIGMOD Record**, Volume 28 Issue 1
Full text available: pdf(1.96 MB) Additional Information: full citation, index terms

18 Informed prefetching and caching
R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka
December 1995 **ACM SIGOPS Operating Systems Review , Proceedings of the fifteenth ACM symposium on Operating systems principles**, Volume 29 Issue 5
Full text available: pdf(2.13 MB) Additional Information: full citation, references, citings, index terms

Results 1 - 18 of 18

Your search matched **1** of **1117580** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevance Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

'orphan elimination in distributed object oriented sys' Search

☐ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

---

1 **Orphan elimination in distributed object-oriented systems**
Ezzat, A.K.;
Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop (Future Trends of , 30 Sept.-2 Oct. 1990
Pages:403 - 412

[Abstract] [PDF Full-Text (528 KB)] **IEEE CNF**

Orphan Elimination in Distributed Object-Oriented Systems

Ahmed K. Ezzat
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94303
ezzat@hplabs.hp.com

Abstract

The orphan detection and elimination issue has been addressed extensively in the distributed transaction management context and more recently in the distributed systems context within the RPC framework. An orphan in a distributed object-oriented applications environment is an object which is created by an application and continues to exist beyond the application's lifetime. This paper addresses orphan detection and elimination for applications in a distributed object-oriented system. We apply our model for a distributed Unix/C++ environment, but the model is general and can be applied to other distributed object-oriented systems.

1 Introduction

A distributed system consists of multiple computers (called hosts) that communicate through a network. A distributed application is one whose components reside and execute at multiple hosts in a distributed system. In a distributed object-oriented application environment, we envision the object-oriented programming language to be the view of the application programmer in a possibly heterogeneous distributed object system.

An orphan in a distributed object-oriented application environment is a non-persistent object which is created by an application and continues to exist beyond the application's lifetime. The orphan detection and elimination issue has been addressed extensively in the distributed transaction management context [Herlihy 87, Herlihy 89, Liskov 87, McKendry 86, Mueller 83, Walker 84] and more recently in the distributed systems context within the RPC framework [Birrell 83, Casey 86, Lampson 81, Panzieri 88, Ravindran 89, Shrivastava 83].

A widely accepted approach in constructing reliable distributed programs is the use of atomic actions (also called transactions). Atomicity encompasses three properties: serializability, recoverability, and permanence of effect. *Serializability* [Gray 78, Bernstein 81, Papadimitriou 79] means that the execution of one activity never appears to overlap or contain the execution of another activity. *Recoverability* means that the overall effect of an activity is all-or-nothing, and *permanence of effect* means that once an action commited its effect will survive failures [Bernstein 83, Attar 84, Gray 81].

In a distributed transaction system, an orphan can be created because of crashes or aborted transactions. Orphans in such an environment are undesirable because they are an activity executing on behalf of an aborted transaction, and they can introduce unnecessary delay or deadlock by holding locks needed by nonorphans. In that sense, it is not only desirable to eliminate orphans, but also it should be done in a timely manner.

There have been many algorithms for orphan elim-

0-8186-2088-9/90/0000/0403/$01.00 © 1990 IEEE

ination in a distributed transaction environment: *eager* algorithms [Herlihy 89, McKendry 86] which use real-time clocks to ensure that orphans are eliminated within a fixed duration, *lazy* algorithms [Herlihy 89, McKendry 86] which use logical clocks to ensure that orphans are eventually eliminated as information propagates through the system, and a *topological change* algorithm suitable for dealing with orphans resulting from network partitioning [Mueller 83] in a distributed transaction system. Mueller's algorithm is driven by both the transaction home sites and the transaction synchronization sites (TSS) for the different files opened from the transaction.

Outside the transaction domain, the orphan elimination problem was first identified by Birrell [Birrell 83], and solutions based on timeouts have been proposed by Lampson [Lampson 81] and Rajdoot [Panzieri 88]. Depending on the RPC semantics in a distributed system, additional requirements might be needed for correctness criteria. In particular, for *at most once* calls, not only should orphans be aborted, the abortion must include undoing of any side effects that may have been produced [Panzieri 88]. Walker [Walker 84] has proposed a transaction-based orphan elimination scheme that dynamically tracks dependencies among transactions. His scheme uses timeouts to keep the information sent in messages to a manageable level.

In a conventional transaction system, orphans have no impact on data consistency, where a transaction does not interact with the outside world until it commits. However, in a general purpose distributed system, such inconsistencies may be more problematic. For example, the Argus system [Liskov 83] supports a methodology in which user-defined atomic data types are implemented by a mixture of atomic and nonatomic data types at a lower level. In the absence of an orphan elimination scheme, the implementor of such a type is responsible for ensuring that transient inconsistencies in the atomic components do not produce permanent inconsistencies in the nonatomic components. An orphan elimination scheme based on Walker's method has been implemented as part of the Argus system [Liskov 87] to deal with that issue.

This paper addresses orphan detection and elimination for applications in a distributed object-oriented systems. We apply our model for a distributed Unix/C++ environment, but the model is general and can be applied to other distributed object-oriented systems. The paper is organized as follows. Section 2 describes the background for this work and gives definitions for the terminology used in the paper. Section 3 presents the problem definition. Section 4 discusses our model. Section 5 uses a simple example in a distributed Unix/C++ environment to describe our model in more detail, and we conclude by a summary in Section 6.

2 Background and terminology

In this section, we define some terms used by our model in this paper.

2.1 Implementation set

An implementation set is an object with a well defined interface representing the traditional Unix process abstraction in an object-oriented system environment. An implementation set contains many objects (persistent and non-persistent), and exports operations to administer these objects as well as control migrating objects between different implementation sets. Implementation sets are uniquely identified in the system by a unique `IS-id`, i.e., corresponding to Unix process-id.

2.2 Distributed thread of control

Traditional threads [Cooper 88] will be referred to as physical threads in this paper as they have their own physical stacks. Typically, a physical thread is uniquely identified within a process by a `thread-id`. Distributed thread (*dthread*) represents the sequence (flow) of events or invocations by an application (one or more physical threads) in a distributed system, i.e., the dthread notion is a virtual thread of control and it does not own private physical stacks, rather it uses potentially many of the stacks belonging to existing physical threads.

Dthreads are created upon invoking operations on objects asynchronously, i.e., an application might have a tree of distributed threads. A distributed thread terminates when the method originally invoked asynchronously terminates.

The notion of dthread is used to help in tracing events and cleaning orphans later. Each dthread has a unique identifier in the system (dthread-id) assigned at creation time in a decentralized manner using the tuple `<machine-id : IS-id : thread-id>`[1].

2.3 Factories

Factories are objects specialized in creating objects of a specific class, i.e., interface and implementation. They provide a uniform model for creating objects in a distributed environment. This model allows the programmer to create objects the same way independently of where they are created (i.e., in the local address space or in a different one).

In traditional C++, the operator `new()` creates an instance of a class only in the same address space. Factories are responsible only for creating objects[2] of a specific class. The interface to a factory object has one method (`create()`), which creates instances of a particular class in the same address space where the factory is. The `create()` method can be thought of as a wrapping to the `new()` operator in C++. Once the programmer has a reference to a factory object, the programmer can create objects of that specific class by invoking the `create()` method on that factory independent of the factory location.

2.4 Persistent objects

Persistent objects are objects whose lifetime is independent of the lifetime of the thread which created them. On the other hand, non-persistent object's lifetime is tied to the lifetime of the thread which created them, for example, in traditional Unix/C++ environment objects are deleted automatically when the process terminates.

3 Problem statement

The current memory management model for C++ is to reclaim memory for all objects in a given process (address space) upon process termination. This model would not apply for persistent objects as by definition a persistent object's lifetime is independent of the lifetime of the dthread which created it.

On the other hand, applying the current C++ memory management model to non-persistent objects is a much more difficult problem and needs to be resolved. For example, consider a dthread that creates a remote non-persistent object and then terminates. This remote object becomes an orphan and needs to be reclaimed. Next section presents a lazy scheme where orphans are

[1] In a Unix context, machine-id corresponds to a host-id, IS-id corresponds to a process-id, and thread-id is a unique identifier to a physical thread within that process.

[2] Notice that factories only create objects and do not delete or manage them. Deleting an object will be dealt with in this paper in a later section.

guaranteed to be eliminated at the termination of the dthread which created them.

4 The model

The issue of managing the application objects' storage is fortunately limited only to two scenarios: an application object created as a result of remote invocation on a factory object, and an object migrating from one implementation set to another. We envision one approach for dealing with both scenarios and in our discussions we will focus only on the factory invocation scenario.

To administer globally the storage of distributed application objects, we are adding two objects per implementation set to help in performing this administrative function. These two objects are either created transparently as part of initializing new implementation sets or explicitly by requiring applications to call a library function from `main()`. Each of the two objects has a database as part of their state representing the information about a subset of the objects in the system, and each object has a set of methods to query their database. These two objects enable tracking where dthreads had created objects and how to get to them. The operation of how they are used is discussed in detail next section through a simple example. The following is a description of these two objects and their interfaces' methods description.

1. **Local directory (LD)** object: this is an object per implementation set. The object's state contains information about all local objects in this implementation set which are created in response to requests from other implementation sets. An entry in this object's database is represented as a tuple `<dthread-id : IS-id : GID : local object reference>`. Figure 1 represents the *local directory* interface declaration. Referring to figure 3, an entry in the local directory object's database in IS2 is described below:

   - Dthread-id: is the identification of the dthread (dthread1) which created this object (*object1*) in this implementation set (IS2).
   - IS-id: is the implementation set (IS1) from which the dthread initiated the creation of this object (object1).
   - GID: is the object (object1) global unique-id.
   - Local object reference: is a reference to the local object (object1) in this implementation set (IS2).

   The `add_entry()` method inserts a tuple in the local directory's state about how a local object in this implementation set was created.

   The different flavors of the `delete_entry()` method delete the appropriate local object(s) in this implementation set and clean their corresponding entries in this local directory object.

2. **Remote directory (RD)** object: this is an object per implementation set. The object's state contains information about remotely created objects from this implementation set. An entry in this object's database is represented as a tuple `<dthread-id : GID : remote LD reference : remote object reference>`. Figure 2 represents the *remote directory* interface declaration. Referring to figure 3, an entry in the remote directory object's database in IS1 is described below:

   - Dthread-id: is the identification of the dthread (dthread1) which created the remote object (object1).
   - GID: is the object (object1) global unique-id.

```
interface local_directory =
          add_entry(dthread_id ID, IS_id IS, gid_t GID, ref_t LCL_OBJ_REF),
          delete_entry(ref_t  LCL_OBJ_REF),
          delete_entry(gid_t  GID),
          delete_entry(dthread_id ID, IS_id IS);
```

Figure 1: **A local directory interface declaration**

```
interface remote_directory =
          add_entry(dthread_id ID, gid_t GID, ref_t RMT_LD_REF, ref_t RMT_OBJ_REF),
          delete_entry(ref_t  RMT_OBJ_REF),
          delete_entries(ref_t  RMT_LD_REF),
          check(dthread_id  ID);
```

Figure 2: **A remote directory interface declaration**

- Remote LD reference: is a remote reference to the local directory object in the implementation set where object1 is (i.e., LD2).
- Remote object reference: is a remote reference to the actual object (i.e., object1) in IS2.

The `add_entry()` method inserts a tuple in the remote directory's state about an object created remotely from this implementation set.

The `delete_entry()` method deletes a remote object and cleans the appropriate directory entries.

The `delete_entries()` method deletes all entries in this remote directory object for objects created from this implementation set in a specific remote implementation set.

The `check()` method returns an entry from the remote directory object corresponding to a specific dthread-id.

A new two library functions are supplied as part of the system, one is to eliminate orphan objects on dthread termination (`terminate()`) and the other is used for explicit object deletion (`kill()`). These library functions will invoke operations on the appropriate directory objects to track where dthreads created objects and how to get to them to delete them. Access to these two directory objects is transparent to the application programmer (i.e., the application need not to do explicit operations on these directory objects) and they are only accessed from system functions. The following are three scenarios illustrating when these directory objects are accessed:

- On creating a remote object, the factory's `create()` method initializes an entry in the local directory object for the implementation set where the factory is. Back on the client side, the `create()` method will initialize an entry in the remote directory object before returning a reference to the application.
- On dthread termination, the `terminate()` library function invokes operations on the appropriate directory objects to track and delete orphans, and then clean the appropriate entries from these directory objects' state.
- On explicit object deletion, the `kill()` library function invokes operations on the appropriate directory objects to track and delete the object (i.e., if the object is remote), and to clean the appropri-

ate entries from the directory objects' state.

5 Example

To illustrate how an implemented version of our model will work in a distributed Unix/C++ environment, we use a simple example (figure 3) where, dthread1 created in IS1 creates (object1) in IS2, and then terminates. In this simple example, we assume that the application has already a reference to *factory1*. The example will discuss in detail how remote objects are created, how orphans are eliminated at dthread termination, and how objects are deleted explicitly.

5.1 Remote object creation

Objects are created by invoking the `create()` method on the appropriate factory. In this example (see figure 3), dthread1 in IS1 invokes the `create()` method on factory1 in IS2 and as a result, object1 is created in IS2, and a reference to the remote object is returned to dthread1. In order to keep track of remote objects, appropriate entries in the directory objects are initialized as follows: The following steps occur transparently to the application:

- The factory in IS2 after creating the desired object (object1) will invoke the `add_entry()` operation on LD2 object. This will add an entry in the local directory object's database in IS2 `<dthread1 : IS1 : GID : *object1>`.
- In the RPC message returning the reference to object1, the factory will add a reference to LD2 in the message.
- On crossing machine boundary to IS1, An `add_entry()` operation is invoked on the remote directory object in IS1. This will add an entry in the remote directory object's database in IS1 `<dthread1 : GID : *LD2 : *object1>`.
- A reference to the remote object `object1` is returned to the applications.

5.2 Dthread termination

In the above example, upon dthread1 termination in IS1, the `terminate()` library function is called to eliminate all non-persistent objects created by dthread1 in the system which otherwise become orphans. The `terminate()` function checks (see figure 4) if there are entries in the remote directory object (RD1) belonging to dthread1. If no entries are found, then dthread1 terminates and we are done.

For every entry found in the RD1 object belonging to dthread1, the `check(ID)` operation returns a reference (`RMT_LD_REF`) corresponding to a local directory object (i.e., `LD2`). The `terminate()` function invokes the `delete_entry(GID)` method on the local directory object in the remote implementation set (IS2). This will result in deleting object1 and the corresponding entry from the local directory object in IS2. With no entries in the remote directory object in IS2 corresponding to dthread1, the `delete_entry()` operation is completed. Finally, the `terminate()` function invokes the `delete_entries()` method on the remote directory object in IS1 to clean all entries with `RMT_LD_REF` field equal to the returned reference from the `check()` operation (i.e., `LD2`).

Notice that the `delete_entry()` operation on the local directory object after completing its work in IS2, it checks if there are entries in the remote directory object at IS2 belonging to dthread1. If there are entries, the `delete_entry()` is invoked on the new local directory object as we discussed earlier. This will take care of nested object creation across multiple implementation

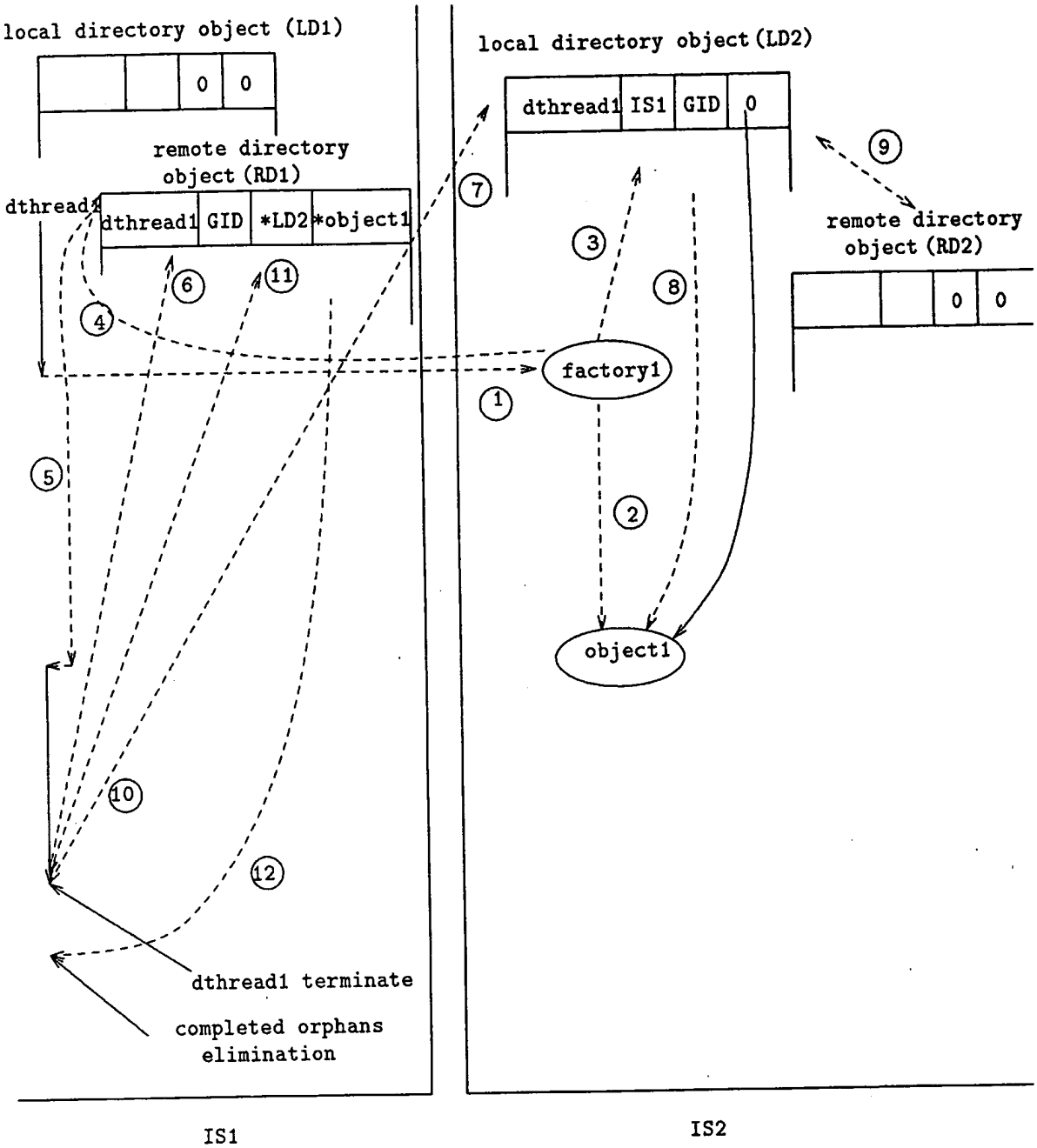

Figure 3: Creating a remote object and orphan reclaimation

```
terminate(dthread_id  ID)
begin

          ref_t    RMT_LD_REF;

          while(RMT_LD_REF := remote_directory->check(ID)){
                RMT_LD_REF->delete_entry(ID, IS);
                remote_directory->delete_entries(RMT_LD_REF);
          }
end;
```

Figure 4: **The terminate library function**

sets in a distributed system.

5.3 Explicit object deletion

Factories are used to provide a uniform method of creating objects in a distributed system. They are responsible for creating objects and not for managing or deleting them. This is because we believe that the programmer should not have to remember the factory reference in order to be able to delete an object. Instead, we introduce a new `kill()` function to replace the `delete()` function in C++.

If the object to be deleted is remote (i.e., dthread1 in IS1 is deleting object1 in IS2, the `kill(obj_ref)` function (see figures 3, 5) invokes the `delete_entry(obj_ref)` method on the remote directory (RD1) in this implementation set (IS1). This in turn invokes the `delete_entry(GID)` method on the appropriate remote LD object (i.e., LD2). The net result is deleting the remote object (object1) and cleaning the appropriate entries in the directory objects, i.e., an entry in the local directory object in IS2 and an entry in the remote directory object in IS1.

On the other hand, if the object is local (e.g., dthread1 in IS2 is deleting object1 and `obj_ref` is a reference to object1), the `kill(obj_ref)` function invokes the `delete_entry(obj_ref)` method on the local directory object in this implementation set (i.e., IS2). The net result is deleting the object (object1) and cleaning the appropriate entry in the local directory object (i.e., IS2).

6 Conclusion

The orphan detection and elimination issue has been investigated extensively in the distributed transaction context as well as in the distributed system context within the RPC framework. In this paper we extended this investigation to a general distributed object-oriented application environment. Orphans in a distributed object-oriented application environment are objects which are created by the application and continue to exist beyond the application's lifetime. Orphan objects in a distributed object-oriented application environment correspond to activities which continue to execute on behalf of aborted transactions in a distributed transaction system.

We presented a simple and a general model for detecting and eliminating orphans in a distributed object-oriented application environment. Also, we presented a simple example to illustrate how our model would work in a distributed Unix/C++ application environment. An implementation of our model should not add much overhead; we are planning to implement this model in

```
void* kill(ref_t obj_ref)
begin
        if (obj_ref->is_remote())
                remote_directory->delete_entry(obj_ref);
        else
                local_directory->delete_entry(obj_ref);

end;
```

Figure 5: The kill function

the near future as soon as the necessary platform environment is completed.

References

[Attar 84] R. Attar, P.A. Bernstein and N. Goodman, "Site Initialization, Recovery, and Backup in a Distributed Database System," *IEEE Trans. on Soft. Eng*, vol.SE-10, No. 6, pp. 645-650, (Nov. 1984).

[Bernstein 81] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, vol. 13, No. 2, pp. 185-221, (June 1981).

[Bernstein 83] P.A. Bernstein, N. Goodman and V. Hadzilacos, "Recovery Algorithms for Database Systems," *Proc. of the IFIP 9th World Computer Congress*, pp. 799-807, (1983).

[Birrell 83] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *Operating Systems Review*, vol. 17, no. 5, (Oct. 1983).

[Casey 86] L.M. Casey, "Comments on "The Design of a Reliable Remote Procedure Call Mechanism"," *IEEE Trans. on Computers*, vol. C-35, No. 3, pp. 283-285, (March 1986).

[Cooper 88] E. Cooper and R. Draves, "C-Threads," Carnegie-Mellon University, Tech. Rep. CMU-CS-88-154, (March 1988).

[Gray 78] J.N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems-an Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmuller, Springer Verlag, New York, pp. 393-481, (1978).

[Gray 81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, vol. 13, No. 2, pp. 223-242, (June 1981).

[Herlihy 87] M.P. Herlihy, N. Lynch, M. Merritt, and W. Weihl, "On the Correctness of Orphan Elimination Algorithms," *Symp. on Fault Tolerant Computing*, pp. 8-13, (1987).

[Herlihy 89] M.P. Herlihy, M.S. McKendry, "Timestamps-Based Orphan Elimination," *IEEE Trans. on Soft. Eng*, vol. 15, No. 7, pp. 825-831, (July 1989).

[Lampson 81] B. Lampson, "Remote Procedure Calls," *Lecture Notes in Computer Science 105*, Berlin: Springer-Verlag, pp. 365-370, (1981).

[Liskov 83] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic support for robust, distributed programs," *ACM Trans. Program. Lang. and Syst.*, vol. 5, No. 3, pp. 381-404, (July 1983).

[Liskov 87] B. Liskov and R. Scheifler, E. Walker, W.E. Weihl, "Orphan detection," *17th Symp. Fault Tolerant Computer Systems (FTCS)*, pp. 2-7, (July 1987).

[McKendry 86] M.S. McKendry and M. Herlihy, "Time-Driven Orphan Elimination," *Fifth Symp. on Relia-*

bility in Distributed Software and Database Systems, pp. 42-, (January 1986).

[Mueller 83] E.T. Mueller, J.D. Moore, and G.J. Popek, "A Nested Transaction Mechanism for LOCUS," *Proc. of the Ninth ACM Symp. on Operating Systems Principles*, pp. 71-85, (October 1983).

[Panzieri 88] F. Panzieri and S. Shrivastava, "Rajdoot: A Remote Procedure Call mechanism Supporting Orphan Detection and Killing," *IEEE Trans. on Soft. Eng*, vol. 14, No. 1, pp. 30-37, (January 1988).

[Papadimitriou 79] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM*, vol. 26, No. 4, pp. 631-653, (October 1979).

[Ravindran 89] K. Ravindran and S.T. Chanson, "Failure Transparency in Remote Procedure Calls," *IEEE Trans. Compt.*, vol. 38, No. 8, pp. 1173-1187, (August 1989).

[Shrivastava 83] S. Shrivastava, "On the Treatment of Orphans in a Distributed System," *Third Symp. on Reliability in Distributed Software and Database Systems*, pp. 155-162, (Nov. 1983).

[Walker 84] E.F. Walker, "Orphan detection in the Argus system," Massachusetts Inst. technol., Lab. Comput. Sci., Tech. Rep. TR-326, (June 1984).

Your search matched **0** of **1117580** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevance Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or enteri new one in the text box.

orphan and thread and priority Search

☐ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

---

Results:
No documents matched your query.